요소 별 영향도의 이해

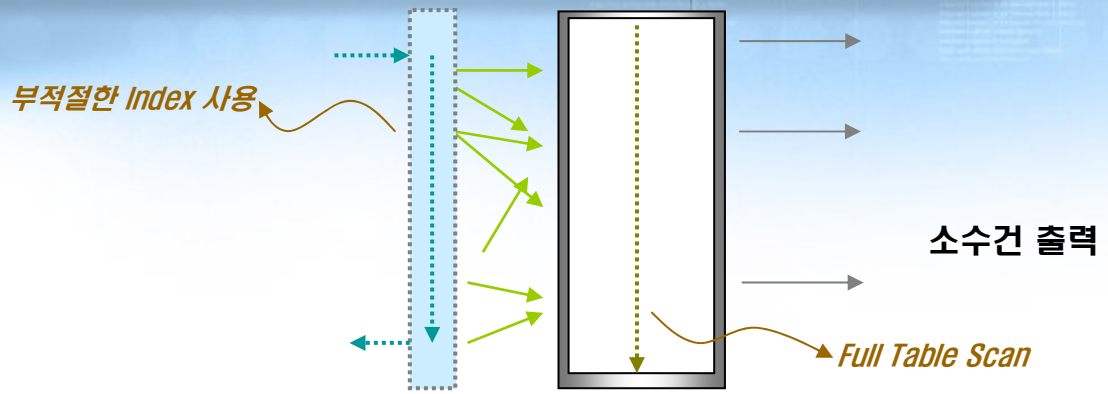
- SQL사용의 영향

SQL 성능개선을 위한 전략적 접근

- 실행계획은 확인하고 있는가?
  - → 옵티마이져가 어떻게 SQL을 수행하려 하는지 그 경향을 파악하라
    - 옵티마이져 모드는? 통계정보는 생성되어 있는가?
    - 옵티마이져의 행동 양식을 유추하자!!!
- SQL의 집합적 처리는 하고 있는가?
- 해당 프로그램으로 인해 영향을 받는 다른 Factor는 없는가?
- 인덱스 전략은 확실하게 수립하였는가?
- 개선의 여지가 더 있는가?

SQL 물리적 수행 순서

WHERE 절에서의 비효율



상황	• 다량의 데이터를 포함한 테이블에서 정상적으로 인덱스를 사용하지 못함으로 인한 비효율 발생 • INDEX를 통해 결과를 최적으로 줄일 수 있음

비효율 원인	개선방안
• 인덱스 컬럼의 변형	• 우측 상수 변형으로 동일 조건 유도(query transforming) • 조건을 구간별 분리 하여 작성(연월의 구간을 년 별로 분리) • 모델 변경(원자화, 타입과 입력 값의 일치) • FBI 사용
• 부정 비교, NULL 비교	• 긍정조건으로 변형, 디폴트 값 사용
• 인덱스 없음	• 인덱스 최적화 전략 수립
• 미사용 또는 비효율적 액세스	• 징검다리 기법 이용
• Binding 변수의 데이터 타입 매칭 오류	• 입력값의 데이터 타입과 컬럼의 데이터 타입 일치
• Static 쿼리 경우 통계정보 미 반영	• 실행계획 분리
• 드라이빙 조건과 필터 조건의 역전	• 역할 전환

인덱스 컬럼 변형 - 1

- 인덱스의 구성 컬럼을 변형하여 INDEX를 사용할 수 없는 경우

  인덱스를 사용할 수 있도록 입력값을 변경

```
S_EMP INDEX : DEPT_ID
DEPT_ID     : NUMBER
입력값      : CHAR
```

```
SELECT *
FROM S_EMP
WHERE TO_CHAR(DEPT_ID) = '50'
```

OR

인덱스 컬럼 변형 - 1

- 인덱스의 구성 컬럼을 변형하여 INDEX를 사용할 수 없는 경우
  인덱스를 사용할 수 있도록 입력값을 변경

```
S_EMP INDEX : DEPT_ID
DEPT_ID     : NUMBER
입력값      : CHAR
```

```
SELECT *
FROM S_EMP
WHERE TO_CHAR(DEPT_ID) = '50'
```

```
SELECT *
FROM S_EMP
WHERE DEPT_ID = TO_NUMBER('50')
```

OR

```
SELECT *
FROM S_EMP
WHERE DEPT_ID = 50
```

인덱스 컬럼 변형 – 2

- 인덱스 컬럼을 결합하여 사용

  인덱스를 사용할 수 있도록 조건절을 분리

```
SELECT  A.YY || A.MM, DEPT, SUM(SALE_QTY)
  FROM  TAB_A A
WHERE   A.YY || A.MM  BETWEEN   '0610'
                          AND   '0704'
 GROUP BY  A.YY || A.MM, DEPT ;
```

TAB_A INDEX: YY + MM + DD + SALE_NO

- 단점 : 범위가 커질 경우 or 조건이 많아지게 되어 문제의 소지가 있음
- 조건입력범위를 알 수 없어 고정형인 static sql이 아닌 literal sql(dynamic)로 작성되어야 할 수도 있음.
- 결합 컬럼을 통한 데이터 모델 변경

==> yyyymm 의 결합 컬럼 생성으로 해결

인덱스 컬럼 변형 – 2

- 인덱스 컬럼을 결합하여 사용

  인덱스를 사용할 수 있도록 조건절을 분리

```
SELECT  A.YY || A.MM, DEPT,
        SUM(SALE_QTY)
  FROM  TAB_A A
WHERE   A.YY || A.MM  BETWEEN   '0610'
                      AND   '0704'
 GROUP BY  A.YY || A.MM, DEPT ;
```

TAB_A INDEX: YY + MM + DD + SALE_NO

- 단점 : 범위가 커질 경우 or 조건이 많아지게 되어 문제의 소지가 있음
- 조건입력범위를 알 수 없어 고정형인 static sql이 아닌 literal sql(dynamic)로 작성되어야 할 수도 있음.
- 결합 컬럼을 통한 데이터 모델 변경

==> yyyymm 의 결합 컬럼 생성으로 해결

```
SELECT  A.YY || A.MM, DEPT, SUM(SALE_QTY)
  FROM  TAB_A A
 WHERE  (A.YY = '06' AND A.MM BETWEEN '10'  AND '12')
     OR  (A.YY = '07' AND A.MM BETWEEN '01'  AND '04')
GROUP BY  A.YY || A.MM, DEPT ;
```

인덱스 컬럼 변형 - 3

- 원자화되지 않은 속성으로 인한 성능 비효율
  - 속성의 원자화를 통하여 인덱스 사용할 수 있도록 조치

```
SELECT  … , columns, …
FROM  TABLE1 x, TABLE2 y
WHERE  x.A || x.B || x.C = y.D
```

TABLE1 INDEX : A + B + C

인덱스 컬럼 변형 - 3

- 원자화되지 않은 속성으로 인한 성능 비효율
  속성의 원자화를 통하여 인덱스 사용할 수 있도록 조치

```
SELECT  … , columns, …
FROM  TABLE1 x, TABLE2 y
WHERE  x.A || x.B || x.C = y.D
```

TABLE1 INDEX : A + B + C

- 인덱스를 사용할 수 있도록 조인되는 상대 테이블의 컬럼을 분화하여 조인
- TABLE1에 A||B||C의 결합 컬럼 생성을 고려
- FUNCTION BASED INDEX(A||B||C)생성 고려

```
SELECT  … , columns, …
FROM  TABLE1,  TABLE2
WHERE  A  =  substr(D, 1, 2)
  AND  B  =  substr(D, 3, 1)
  AND  C  =  substr(D, 4, 3)
```

인덱스 컬럼 변형 - 4

- Index 구성 컬럼의 가공으로 인한 성능 저하.
- Application의 변경도 함께 고려해야 함.

```
select HOLDING_KEY,trim(REC_KEY),
       trim(NBK_MANAGE_KEY),
       ...
 from se_species_tbl
where
   to_char(update_date,'YYYY/MM/DD') =
    to_char(sysdate-1,'YYYY/MM/DD')
or  to_char(create_date,'YYYY/MM/DD') =
    to_char(sysdate-1,'YYYY/MM/DD')
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.03	0.04	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	56	10.34	53.13	61607	95030	4	55
total	58	10.37	53.17	61607	95030	4	55

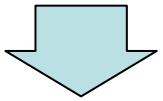
Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
53	TABLE ACCESS (FULL) OF 'SE_SPECIES_TBL'

Call	Count	CPU Time	Elapsed Time	Disk	Query	Current	Rows
Parse	1	0.02	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	0.08	0.90	90	143	0	53
Total	3	0.10	0.92	90	143	0	53

Rows	Row Source Operation
0	STATEMENT
53	CONCATENATION
0	TABLE ACCESS BY INDEX ROWID SE_SPECIES_TBL
1	INDEX RANGE SCAN OF SE_SPECIES_CREATE_DATE_IDX (NONUNIQUE)
53	TABLE ACCESS BY INDEX ROWID SE_SPECIES_TBL
54	INDEX RANGE SCAN OF SE_SPECIES_UPDATE_IDX (NONUNIQUE)

인덱스 컬럼 변형 - 4

```
select HOLDING_KEY,trim(REC_KEY),
       trim(NBK_MANAGE_KEY),
       …
 from se_species_tbl
where
   to_char(update_date,'YYYY/MM/DD') =
    to_char(sysdate-1,'YYYY/MM/DD')
or  to_char(create_date,'YYYY/MM/DD') =
    to_char(sysdate-1,'YYYY/MM/DD')
```



```
select HOLDING_KEY, trim(REC_KEY),
       trim(NBK_MANAGE_KEY),
       …
 From se_species_tbl
where update_date   = trunc(sysdate-1)
    or create_date = trunc(sysdate-1)
```

- Index 구성 컬럼의 가공으로 인한 성능 저하.
- Application의 변경도 함께 고려해야 함.

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.03	0.04	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	56	10.34	53.13	61607	95030	4	55
total	58	10.37	53.17	61607	95030	4	55

Rows	Execution Plan
0	SELECT STATEMENT GOAL: CHOOSE
53	TABLE ACCESS (FULL) OF 'SE_SPECIES_TBL'

Call	Count	CPU Time	Elapsed Time	Disk	Query	Current	Rows
Parse	1	0.02	0.02	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	0.08	0.90	90	143	0	53
Total	3	0.10	0.92	90	143	0	53

Rows	Row Source Operation
0	STATEMENT
53	CONCATENATION
0	TABLE ACCESS BY INDEX ROWID SE_SPECIES_TBL
1	INDEX RANGE SCAN OF SE_SPECIES_CREATE_DATE_IDX (NONUNIQUE)
53	TABLE ACCESS BY INDEX ROWID SE_SPECIES_TBL
54	INDEX RANGE SCAN OF SE_SPECIES_UPDATE_IDX (NONUNIQUE)

To_char(update_date,'yyyy/mm/dd'), to_char(create_date,'yyyy/mm/dd') 로 FBI 생성도 고려해볼 것

인덱스 컬럼 변형 - 5

- Index 구성 컬럼의 가공으로 인한 성능 저하
- FBI 생성 OR INDEX활용할 수 있는 SQL로 전환

```
SELECT COUNT(1)
FROM   TC_CD_INFO
WHERE SUBSTR(UP_CD_ID,1,5) = '21104' AND CD_NM LIKE '%'||:B1 ||'%'
```

```
SELECT COUNT(1)
FROM   TC_CD_INFO
WHERE UP_CD_ID like  '21104'||'%' AND CD_NM LIKE '%'||:B1 ||'%'
```

```
Optimizer mode: ALL_ROWS
Parsing user id: 86  (NSI_IN)   (recursive depth: 1)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   MODE: ALL_ROWS
      0   SORT (AGGREGATE)
      0    TABLE ACCESS   MODE: ANALYZED (FULL) OF 'TC_CD_INFO' (TABLE)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	5	0.00	0.00	0	0	0	0
Execute	5	0.00	0.00	0	0	0	0
Fetch	5	11.89	39.03	78788	230600	0	5
total	15	11.89	39.04	78788	230600	0	5

INDEX 미사용

- 인덱스 구성 선두 컬럼에 조회조건이 입력되지 않았을 경우
  : 징검다리 기법을 이용, SKIP SCAN 활용도 가능

index : col1 + col2 , col1에는 (1,2,3)만 있을 경우

```
SELECT *
FROM TABLE
WHERE COL2 = :V1
(FULL SCAN OR SKIP SCAN)
```

```
SELECT *
FROM   TABLE
WHERE  COL1 IN (1,2,3)
AND    COL2 = :V1
```

```
SELECT *
FROM   TABLE
WHERE  COL1 >= 1
AND    COL2 = :V1
```

하지만, 인덱스를 사용하는 것만이 최선은 아님을 반드시 기억하여야 할 것 !!!

Static SQL의 실행계획 분리

- static sql의 통계정보활용도에 대한 제약으로 인해 적절한 실행계획 수립이 어려움
- 입력데이터의 성격과 패턴을 적절히 분리한 실행계획 제어로 해결

```
SELECT *
FROM    CHULGOT
WHERE CHULDATE LIKE :chuldate||'%'
AND   CUSTNO   LIKE :custno||'%';
```

- 평균적인 통계정보만으로 실행계획 수립
- chuldate vs custno의 입력패턴을 유추할 수 없음

Static SQL의 실행계획 분리

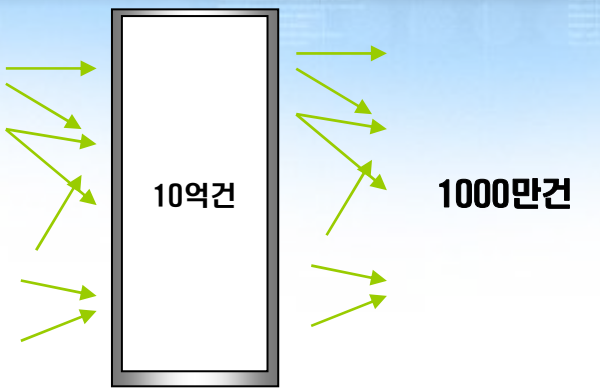
- static sql의 통계정보활용도에 대한 제약으로 인해 적절한 실행계획 수립이 어려움
- 입력데이터의 성격과 패턴을 적절히 분리한 실행계획 제어로 해결

```
SELECT *
FROM   CHULGOT
WHERE CHULDATE LIKE :chuldate||'%'
AND    CUSTNO   LIKE :custno||'%';
```

- 평균적인 통계정보만으로 실행계획 수립
- chuldate vs custno의 입력패턴을 유추할 수 없음
- 데이터 분포도를 알고 있으나 static으로 구현 하여야 할 경우에도 실행계획 분리가 필요함

```
SELECT *
FROM    CHULGOT
WHERE CHULDATE LIKE :chuldate||'%'
AND   :custno is null   AND :chuldate is not null
Union all
SELECT *
FROM    CHULGOT
WHERE CUSTNO LIKE :custno||'%'
AND   :custno is not null   AND :chuldate is null
Union all
SELECT *
FROM    CHULGOT
WHERE :custno is null   AND :chuldate is null
Union all
SELECT *
FROM    CHULGOT
WHERE CHULDATE LIKE :chuldate||'%'
AND   CUSTNO = :custno
AND   :custno is not null
AND :chuldate is not null
```

SQL 비효율 진단 절차



상황	● 처리 대상이 다수이며 인덱스 상황의 "임계치"를 벗어난 경우 ● 다량의 데이터를 처리해야 할 경우 ● 동일 성격의 데이터가 모여있을 수 있을 때

비효율 원인	개선방안
• 대용량 단일 파티션	• 파티션 활용하여 partition full scan으로 유도

상황	전체로우 ->다수 , 예상 결과 로우->소수 , 결과 로우->다수

비효율 원인	개선방안
• 비효율 인덱스 선택	• 인덱스 최적화 전략 수립 • INDEX(T IDX) 힌트로 최적의 인덱스 사용 고정
• 인덱스 중간 컬럼 조건 누락	• 인덱스 최적화 전략 수립 • 징검다리 기법 이용

비효율적 인덱스 선택

- 인덱스에 대한 최적의 설계가 이루어지지 않아 인덱스에서 다수의 데이터를 선택한 후 테이블에서 Filtering함으로 불필요한 RANDOM I/O 발생
  : INDEX 재설계가 절실히 필요함

```
SELECT   A.Q_NO ,A.Q_STAT ....
FROM  TBL_Q A, TBL_AN B
WHERE A.Q_NO    = B.Q_NO(+)
  AND A.R_DT   BETWEEN  TO_DATE(:F_DT,'YYYYMMDD')
                  AND    TO_DATE(:T_DT,'YYYYMMDD')+1
  AND A.USE_YN    = 'Y'
  AND A.Q_STAT IN ('15', '22', '27')
  AND A.M_T  IN ('T0106' ,'T0111','T0112')
  AND ......
  AND ROWNUM <= 30
```

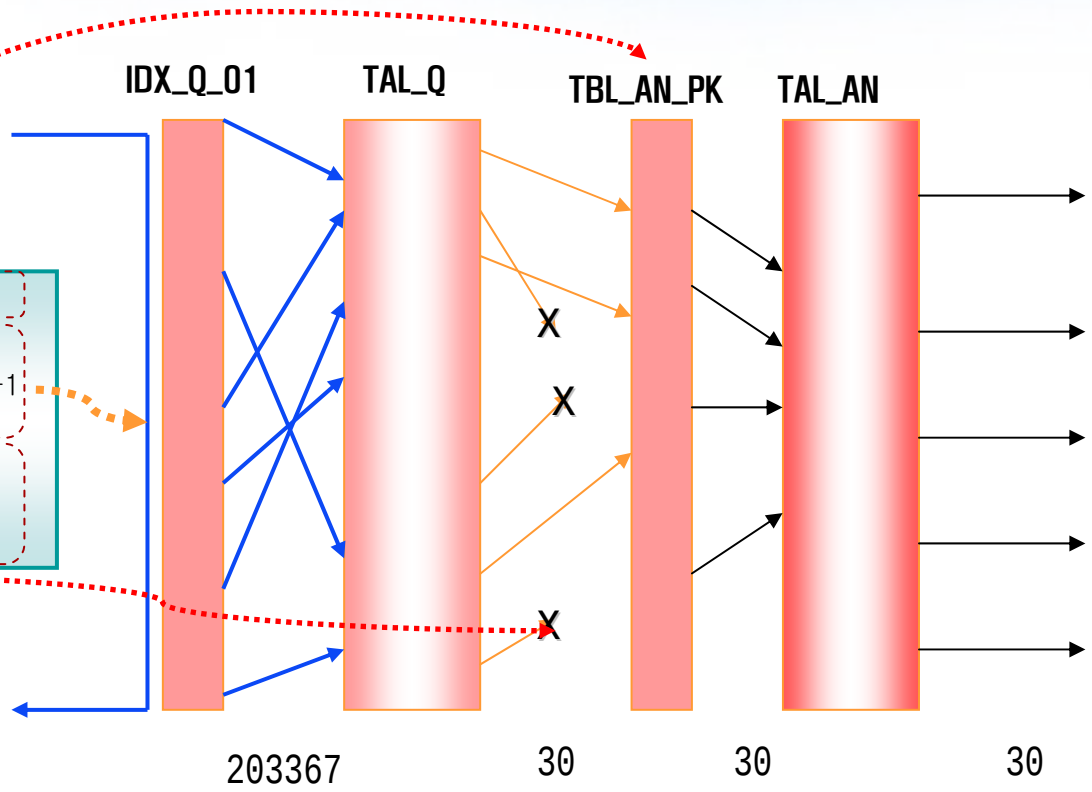
TBL_Q 테이블 인덱스 구성
IDX_Q_01 : R_DT + USE_YN
TBL_AN 테이블 인덱스 구성
TBL_AN_PK : Q_NO+AN_SEQNO

```
call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
Parse        1      0.00       0.01           0           0           0           0
Execute      1      0.00       0.00           0           0           0           0
Fetch        1      3.42      69.39       33861      562110           0          30
-------  ------  --------  ----------  ----------  ----------  ----------  ----------
total        3      3.42      69.39       33419      562110           0          30

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
     30   COUNT STOPKEY
     30    FILTER
     30     NESTED LOOPS OUTER
     30      TABLE ACCESS BY INDEX ROWID TBL_Q
 203367       INDEX RANGE SCAN DESCENDING IDX_Q_01
     30      TABLE ACCESS BY INDEX ROWID TBL_AN
     30       INDEX UNIQUE SCAN TBL_AN_PK
```

비효율적 인덱스 선택

- 인덱스에 대한 최적의 설계가 이루어지지 않아 인덱스에서 다수의 데이터를 선택한 후 테이블에서 Filtering함으로 불필요한 RANDOM I/O 발생
  : INDEX 재설계가 절실히 필요함

```
WHERE A.Q_NO     = B.Q_NO(+)
  AND A.R_DT    BETWEEN  TO_DATE(:F_DT,'YYYYMMDD')
                    AND     TO_DATE(:T_DT,'YYYYMMDD')+1
  AND A.USE_YN     = 'Y'
  AND A.Q_STAT IN ('15', '22', '27')
  AND A.M_T   IN ('T0106' ,'T0111','T0112')
  AND ROWNUM <= 30
```



비효율적 인덱스 선택

- 인덱스에 대한 최적의 설계가 이루어지지 않아 인덱스에서 다수의 데이터를 선택한 후 테이블에서 Filtering함으로 불필요한 RANDOM I/O 발생
  : INDEX 재설계가 절실히 필요함

- R_DT+USE_YN 조건으로 INDEX를 경유하여 Table Access 한 후 다른 조건에 의해 대부분 FILTERING
- IDX_Q_01 인덱스를 재구성 필요
  - Q_STAT 컬럼이 R_DT 컬럼과 함께 사용 되었을 때 가장 변별력이 있음
  - R_DT + Q_STAT+USE_YN 혹은 R_DT + USE_YN + Q_STAT 의 결합인덱스 생성

```
...
      30    FILTER
      30     NESTED LOOPS OUTER
      30      TABLE ACCESS BY INDEX ROWID
  203367       INDEX RANGE SCAN DESCENDIN
      30      TABLE ACCESS BY INDEX ROWID
      30       INDEX UNIQUE SCAN TBL_AN_P
```

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  STATEMENT
     30   COUNT STOPKEY
     30    FILTER
     30     NESTED LOOPS OUTER
     30      TABLE ACCESS BY INDEX ROWID TBL_Q
     56       INDEX RANGE SCAN DESCENDING IDX_Q_01
     30      TABLE ACCESS BY INDEX ROWID TBL_AN
     30       INDEX UNIQUE SCAN TBL_AN_PK
```

비효율적 인덱스 선택

- 최적의 성능을 보장하는 INDEX가 존재함에도 사용하지 못할 때
- 최적의 인덱스를 사용할 수 있도록 힌트로 강제하기

```
CREATE INDEX IDX_TEST_1 ON IDX_TEST (ID);
CREATE INDEX IDX_TEST_2 ON IDX_TEST (NAME) ;

SELECT COUNT(*)
FROM IDX_TEST
WHERE ID BETWEEN 1 AND 100  : 100건

SELECT COUNT(*)
FROM IDX_TEST
WHERE NAME LIKE 'A%'  : 104105 건
```

- 힌트는 최적이라 생각될 때만 사용할 것
  - 과유불급(過猶不及)
- CBO환경에서 똑똑한 통계정보가 있을 경우 판단 MISS하지 않는다 → 똑똑한 판단할 수 있도록 도와줄 것
- !!!. 만약 힌트에 사용한 index에 비효율이 발생할 경우 예상하지 못한 결과를 초래할 수 있음

```
SELECT *
FROM   IDX_TEST
WHERE  ID >= 1 AND ID <=100
AND    NAME LIKE 'A%'  (rule 우선순위 높음)
```

```
SELECT /*+ INDEX(IDX_TEST IDX_TEST_1) */ *
FROM   IDX_TEST
WHERE  ID >= 1 AND ID <=100
AND    NAME LIKE 'A%'
```

비효율적 인덱스 선택

- 최적의 인덱스 선정 및 인덱스 컬럼 구성은 물리 설계 단계에서의 핵심적인 옵티마이징 전략임.
- 적절한 인덱스 구성은 시스템 응답 속도에 직접적인 영향을 주는 가장 중요한 요소의 하나임.

[Table1 상황] Columns 수 : 115개, Index1 (col7), Index2 (col1), Index3 (col8), Index4 (col9)

```
SELECT  A.col1, …
  FROM  Table1 A, Table2 B
WHERE  B.col10 = A.col6  AND  B.col11 = :v2
    AND   A.col1 = :v1  AND  A.col4  = :v3
    AND   A.col5 >= TO_DATE(:v4,'YYYYMMDD')
    AND   A.col5 <= TO_DATE(:v5,'YYYYMMDD')
ORDER BY A.col1, A.col5
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	3.37	12.26	5225	41444	0	3
total	3	3.38	12.27	5225	41444	0	3

```
Rows   Execution Plan
-----  ---------------------------------------------
    0  SELECT STATEMENT   GOAL: CHOOSE
    3   SORT (ORDER BY)
    3    NESTED LOOPS
36038     TABLE ACCESS (BY INDEX ROWID) OF 'TABLE1'
36039      INDEX (RANGE SCAN) OF 'Index2' (NON-UNIQUE)
    3     INDEX (UNIQUE SCAN) OF  'TABLE2_PK' (UNIQUE)
```

Index_New : col1 + col5

```
Rows  Execution Plan
-----  ---------------------------------------------
    0  SELECT STATEMENT   GOAL: CHOOSE
    3   NESTED LOOPS
    3    TABLE ACCESS (BY INDEX ROWID) OF 'TABLE1'
    4     INDEX (RANGE SCAN) OF 'INDEX_NEW' (NON-UNIQUE)
    3   INDEX (UNIQUE SCAN) OF 'TABLE2_PK' (UNIQUE)
```

비효율적 인덱스 선택

최적의 인덱스 설계의 필요성

```
SELECT  A.검사구분코드,  B.접수일자, B.판매코드, A.검사결과번호,
        A.검사일자,  A.입고일자
  FROM  검사결과 A, 검사접수 B
 WHERE  ( A.검사일자 BETWEEN '20070502' AND '20070503'
   AND  A.검사구분코드='330')
   AND  ( A.입고일자 BETWEEN '20070501'
                         AND '20070502'
     OR 채널대분류 = '10')
   AND    A.접수번호 = B.접수번호
```

- [검사결과 테이블]
- IDX_검사결과_01: 판매코드
- IDX_검사결과_02: 검사구분코드+검사결과번호
- IDX_검사결과_03: 검사구분코드+발주번호+판매코드
- IDX_검사결과_04: 검사구분코드+판매코드+발주번호
- IDX_검사결과_05: 검사구분코드+접수번호+판매코드
- IDX_검사결과_PK: 검사결과번호

검사일이 2007년 5월 2일부터
5월3일에 발생한 입고검사건
(검사구분코드=330) 인 건들 중에서
판매채널이 '인터넷'이거나
입고일자가 2007년 5월 1일에서
2일까지 발생한 검사 건들의
검사접수일자를 알고자 한다.

```
call     count       cpu    elapsed       disk      query    current        rows
-------  ------  --------  ---------  ---------  ---------  ---------  --------
Parse        1      0.00       0.00          0          0          0         0
Execute      1      0.00       0.00          0          0          0         0
Fetch        2      2.43      12.43      38575      38575          0         7
-------  ------  --------  ---------  ---------  ---------  ---------  --------
total        4      2.44      12.99      38575      38575          0         7

Rows     Row Source Operation
-------  ---------------------------------------------------
      7  FILTER
      7   NESTED LOOPS
      7    TABLE ACCESS FULL 검사결과
      7    TABLE ACCESS BY INDEX ROWID 검사접수
      7     INDEX UNIQUE SCAN PK_검사접수
```

비효율적 인덱스 선택

- 인덱스(IDX_검사결과_06: 검사구분코드+ 검사일자 추가

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         29          0           7
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.00       0.00          0         29          0           7


Rows     Row Source Operation
-------  ---------------------------------------------------
      7  FILTER
      7   NESTED LOOPS
      7    TABLE ACCESS BY INDEX ROWID 검사결과
     45     INDEX RANGE SCAN IDX_검사결과_06
      7    TABLE ACCESS BY INDEX ROWID 검사접수
      7     INDEX UNIQUE SCAN PK_검사접수
```

JOIN 구문의 비효율

상황	조인 비용 과다 (COST 비용 과다)
비효율 원인	**개선방안**
• 조인될 테이블 인덱스 부재로 인한 연결고리 이상	• 인덱스 최적화 전략 수립 • 조인 방향 변경 – suppressing, 인덱스 힌트
• 조인 결과 량 다수인데 nested loops join 사용	• 조인방법 변경 – hash, sort merge
• 전체 초기 조인 량 과다	• DRIVING 결과 소->대 순서로 조인 순서 변경 • Star join 고려
• 확장 Join 후에 grouping	• 조인 전에 grouping 적용
• 1:M 인 경우 컬럼, 집합 확장이 아닌 비교를 위해 조인 사용	• Exists 혹은 in 서브쿼리 사용
• Outer 조인에 의한 조인방향 고정 및 연결고리 이상	• 인덱스 최적화 전략 수립 • 모델 통합 – 수직분할 통합
• 참조무결성 오류에 의한 카테시안 조인	• Union, group by 활용
• 세미조인 시 연결 고리 이상 혹은 통계오류로 인한 확인자형 조인	• 인덱스 최적화 전략 수립으로 제공자 형으로 변경 • Suppressing 혹은 힌트 사용

JOIN 비효율

- 통계정보 오류에 의한 JOIN METHOD 선택 오류
  - 초기에는 데이터가 많지 않다는 통계정보에 의해 Index보다는 Full Scan이 더 비용이 적은 것으로 판단
  - 통계정보의 갱신, HINT등으로 제어

아래와 같은 결과가 나타난 원인에 대해 설명해 보시오.

사례 SQL 및 실행계획

4.5초

```
SELECT  A.CUST_N, A.PNT_CLAS_CD, A.PRCS_D, A.PNT_AMT, A.RSN_CD, ...
  FROM   TB4010 A, TC4016 B
WHERE   A.PRCS_D BETWEEN :S004 AND :S005 AND A.PNT_CLAS_CD = :S006
    AND   A.CUST_N = B.CUST_N
```

PK인덱스 컬럼

```
Rows     Execution Plan
--------  ---------------------------------------------------
       0  SELECT STATEMENT   GOAL: CHOOSE
      12    HASH JOIN
      12      TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TB4010'
      13        INDEX   GOAL: ANALYZED (RANGE SCAN) OF 'TB4010_IX1' (NON-UNIQUE)
  770681      TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TC4016'
```

문제점 및 원인

- TC4016의 CUST_N (고객번호)를 선두로 하는 인덱스 미사용
- 불완전한 통계정보에 기인 (매우 오래된 통계정보)
- Hash Join에 의한 Full Table Scan

개선 방안

- TC0416의 PK 인덱스를 사용하고
- Nested Loops 방식으로 조인하도록 Hint로 유도
- 예상 : 0.2초 이내

연결고리 이상으로 인한 JOIN비효율

- 연결대상 테이블에 연결고리가 되는 컬럼의 인덱스가 정상이 아니거나 없을 경우 JOIN ORDER, JOIN METHOD가 잘 못 선택될 수 있음
  - 인덱스 설계전략에 맞추어 인덱스 생성으로 비효율 제거

```
SELECT   A.ORD_DT, A.ORD_CD, MAX (A.CUST_NO) CUST_NO,
         MAX (B.CUST_NM) custnm, MAX (A.ORD_CD) ORD_CD,
         SUM (A.ORD_AMT) ordamt, MAX (A.CONFIG) CONFIG
FROM   TEST_A  A, TEST_B  B
WHERE SUBSTR (A.BOK_CD, 1, 2) = B.SUB_CD
 AND   A.CUST_NO = B.CUST_NO
 AND   A.CONFIG IN ('0', '1')
 AND   A.BOK_CD LIKE :B1' || '%'
GROUP BY A.ORD_DT, A.ORD_CD
ORDER BY A.ORD_DT
```

TEST_A 테이블 CONFIG 컬럼의 분포도
0,1 : 10건 : (- 주로 사용하는 조건임)
그 외 : 100만건

TEST_A INDEX 상황
PK : ORD_DT + CUST_NO + BOK_CD
IDX1 : CUST_NO

TEST_B INDEX 상황
PK : CUST_NO + SUB_CD

```
Rows       Execution Plan
--------   ---------------------------------------------------
         0  SELECT STATEMENT   GOAL: CHOOSE
         5   SORT (GROUP BY)
        20    TABLE ACCESS (BY INDEX ROWID) OF 'TEST_A'
   1142894     NESTED LOOPS
      4958       TABLE ACCESS (FULL) OF 'TEST_B'
   1137935       INDEX (RANGE SCAN) OF 'TEST_A_IDX1' (NON-UNIQUE)
```

연결고리 이상으로 인한 JOIN비효율

[개선방안]

1.옵티마이져는 상수(또는 바인드변수)가 들어오는 컬럼에 인덱스 존재 여부 체크
- TEST_A.CONFG IN ('0', '1') TEST_A.CONFIG 을 선두로 하는 인덱스 부재 및 로우 수 다수

2. 조인 컬럼에 인덱스가 존재하는지 조사 후 Nested Loops 조인/ Hash 조인/ Sort Merge 조인 결정

3. 조인 컬럼에 인덱스가 존재하므로 옵티마이져는 Nested Loops 조인 결정
(TEST_B -> TEST_A_IDX_1 -> TEST_A)

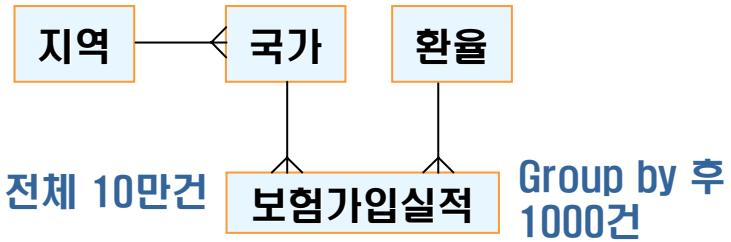
- TEST_A 에 CONFIG+BOK_CD 로 하는 결합 인덱스로 성능 향상

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      5  SORT (GROUP BY)
     20   NESTED LOOPS
            INLIST ITERATOR
     20      TABLE ACCESS BY INDEX ROWID TEST_A
     20        INDEX RANGE SCAN TEST_A_IDX_2
     20     TABLE ACCESS BY INDEX ROWID TEST_B
     20      INDEX UNIQUE SCAN PK_TEST_B
```

확장 JOIN 후 GROUPING

- JOIN은 곱(PRODUCT)
- JOIN으로 집합의 LEVEL이 변경된 후(1 →) 다시 GROUPING으로 LEVEL 변경(M→)
  - M 집합을 미리 GROUPING한 후 1 집합과 조인

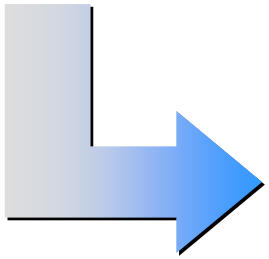
```
SELECT  Y.지역, SUBSTR(종목,1,1),
        SUM(DECODE(GREATEST(가입일,'199312'),'199312',
            외화*DECODE(SUBSTR(종목,1,1),'1',0.25,1)*Z.기준율)) ,
        ..................................................... ,
        SUM(DECODE(SUBSTR(가입일,1,4), '1998',
            외화*DECODE(SUBSTR(종목,1,1),'1',0.25,1)*Z.기준율))
FROM    보험가입실적 X, 국가 Y, 환율 Z
WHERE   Y.국가코드 = X.국가코드
 AND    Z.적용일자 = TO_CHAR(X.가입일,1,4)||'1231'
 AND    Z.통화코드  = Y.통화코드
 AND   X.가입일자 BETWEEN '199001' AND '199812'
GROUP BY  Y.지역, SUBSTR(종목,1,1)
```



확장 JOIN 후 GROUPING

- JOIN은 곱(PRODUCT)
- JOIN으로 집합의 LEVEL이 변경된 후(1 →) 다시 GROUPING으로 LEVEL 변경(M→)
  - M 집합을 미리 GROUPING한 후 1 집합과 조인

```
SELECT  Y.지역, SUBSTR(종목,1,1),
        SUM(DECODE(GREATEST(가입일,'199312'),'199312',
            외화*DECODE(SUBSTR(종목,1,1),'1',0.25,1)*Z.기준율)) ,
        ..... ,
        SUM(DECODE(SUBSTR(가입일,1,4), '1998',
            외화*DECODE(SUBSTR(종목,1,1),'1',0.25,1)*Z.기준율))
FROM    보험가입실적 X, 국가 Y, 환율 Z
WHERE   Y.국가코드 = X.국가코드
 AND    Z.적용일자 = TO_CHAR(X.가입일,1,4)||'1231'
 AND    Z.통화코드  = Y.통화코드
 AND   X.가입일자 BETWEEN '199001' AND '199812'
GROUP BY  Y.지역, SUBSTR(종목,1,1)
```

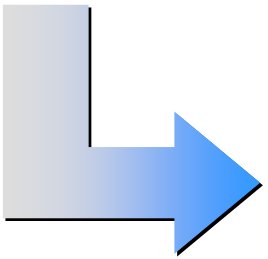


```
SELECT  Y.지역, X.구분,
SUM(DECODE(GREATEST(년,'1993'),'1993',금액*DECODE(구분,'1',0.25,1)*기준율))
SUM(DECODE(년, '1994', 금액*DECODE(구분,'1',0.25,1)*기준율)) ,
  ..... ,
 SUM(DECODE(년, '1998', 금액*DECODE(구분,'1',0.25,1)*기준율))
FROM ( SELECT 국가, SUBSTR(종목,1,1) 구분, SUBSTR(가입일자,1,4) 년,
          SUM(외화) 금액
       FROM    보험가입실적
       WHERE 가입일 BETWEEN '199001' AND '199812'
      GROUP BY 국가, SUBSTR(종목,1,1), SUBSTR(가입일,1,4) ) X, 국가 Y,
       환율 Z
WHERE  Y.국가코드 = X.국가코드
  AND   Z.적용일자 = 년||'1231'
  AND   Z.통화코드  = Y.통화코드
GROUP BY  Y.지역, 구분 ;
```

집합확장이 아닌 단순비교를 위한 조인

- 집합 비교(MINUS,INTERSECT,ETC)를 위해 JOIN할 경우 불필요한 집합 LEVEL변화가 발생
  - IN, EXISTS와 같이 집합의 요건을 정의할 SEMI JOIN으로

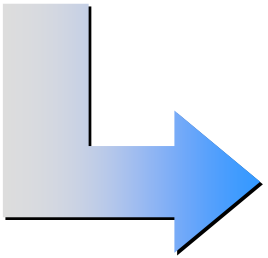
```
SELECT 고객번호, 고객명, 연락처, .............
FROM ( SELECT X.고객번호,
              MAX(X.고객명) 고객명,
              MAX(X.연락처) 연락처,
              .......
         FROM   고객 X,  청구 Y.
         WHERE  X.고객번호 = Y.고객번호
         AND    X.고객상태 = '연체'
         AND    Y.납입구분 = 'N'
       GROUP  BY  X.고객번호
       HAVING SUM(Y.미납금) BETWEEN
              :VAL1 AND :VAL2)
WHERE  ROWNUM <= 2000 ;
```



집합확장이 아닌 단순비교를 위한 조인

- 집합 비교(MINUS,INTERSECT,ETC)를 위해 JOIN할 경우 불필요한 집합 LEVEL변화가 발생
  - IN, EXISTS와 같이 집합의 요건을 정의할 SEMI JOIN으로

```
SELECT 고객번호, 고객명, 연락처, .............
FROM ( SELECT X.고객번호,
              MAX(X.고객명) 고객명,
              MAX(X.연락처) 연락처,
              .......
       FROM   고객 X,  청구 Y.
       WHERE  X.고객번호 = Y.고객번호
       AND    X.고객상태 = '연체'
       AND    Y.납입구분 = 'N'
       GROUP  BY  X.고객번호
       HAVING SUM(Y.미납금) BETWEEN
                  :VAL1 AND :VAL2)
WHERE  ROWNUM <= 2000 ;
```

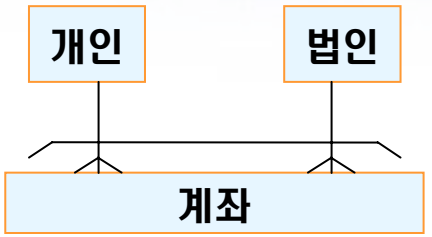


```
SELECT 고객번호, 고객명, 연락처, .............
FROM   고객 X
WHERE 고객상태 = '연체'
AND  EXISTS ( SELECT ' '
              FROM  청구 Y
              WHERE  Y.고객번호 = X.고객번호
              AND    Y.납입구분 = 'N'
              GROUP  BY  Y.고객번호
              HAVING SUM(Y.미납금)
                       BETWEEN :VAL1 AND :VAL2)
AND  ROWNUM <= 2000 ;
```

참조무결성 오류에 의한 카테시안 JOIN

- OUTER JOIN과 함께 참조무결성 위반으로 원하는 집합의 LEVEL이 도출되지 않음
  - UNION → GROUP BY 활용하여 집합 간의 정의를 명확히 하라

```
SELECT x.계좌번호, x.개설일자,
       nvl(y.성명,z.법인명), ……
FROM    계좌 x, 개인 y, 법인 z
WHERE (x.구분 = '1' AND x.ID = y.ID  OR
       x.구분 = '2' AND x.ID = z.ID)
AND    x.개설일 LIKE :in_date||'%'
```




```
SELECT x.계좌번호, x.개설일자,
       nvl(y.성명,z.법인명), ……
FROM    계좌 x, 개인 y, 법인 z
WHERE  y.ID(+) = decode(x.구분, '1', x.ID)
  AND  z.ID(+) = decode(x.구분, '2', x.ID)
  AND  x.개설일자 LIKE :in_date||'%' ;
```

- 항상 3 테이블의 조인이 수행
- 비교 값이 NULL인 경우도 조인은 수행됨

참조무결성 오류에 의한 카테시안 JOIN

- OUTER JOIN과 함께 참조무결성 위반으로 원하는 집합의 LEVEL이 도출되지 않음
  - UNION → GROUP BY 활용하여 집합 간의 정의를 명확히 하라

```
SELECT x.계좌번호, x.개설일자,
       nvl(y.성명,z.법인명), ……
FROM     계좌 x, 개인 y, 법인 z
WHERE  y.ID(+) = decode(x.구분, '1', x.ID)
  AND  z.ID(+) = decode(x.구분, '2', x.ID)
  AND  x.개설일자 LIKE :in_date||'%' ;
```

```
SELECT x.계좌번호, x.개설일자,
       y.성명, ……
FROM     계좌 x, 개인 y
WHERE  x.구분 = '1' AND x.ID = y.ID
  AND  x.개설일자 LIKE :in_date||'%'
UNION ALL
SELECT  x.계좌번호, x.개설일자,
        z.법인명, ……
FROM     계좌 x, 법인 z
WHERE  x.구분 = '2' AND x.ID = z.ID
  AND  x.개설일자 LIKE :in_date||'%' ;
```

- 배타적 관계가 많을 때 코딩량 증가

SEMI JOIN 방식 선택 오류

- 제공자, 확인자형 SEMI JOIN(SUB-QUERY)의 특징에 맞는 실행계획이 수립되어야 함
  - 실행계획을 확인한 후 JOIN METHOD를 정확히 선택하여 실행할 수 있도록 하라!!!
  - 확인자는 확인자로, 제공자는 제공자로...

```
SELECT YYMM ,CLOSE_FLAG ,NVL(PAY_2001_CNT,0)
       ,NVL(PAY_2002_CNT,0)
       ,NVL(PAY_2003_CNT,0) ,NVL(PAY_2004_CNT,0)
       ,NVL(PAY_2004_SAT,0) ,
       ....
FROM  DUTY.BPD18D
WHERE YYMM = :B1
  AND EMP_ID = :B2
  AND CLOSE_FLAG =
```

```
(SELECT MAX(CLOSE_FLAG)
   FROM DUTY.BPD18D
  WHERE EMP_ID = :B2
    AND YYMM = :B1)
```

FILTER

- 서브쿼리가 독립적으로 수행되어 제공될 수 있었음에도 확인자형(FILTER)으로 실행되어 부하발생
- BPD18D의 인덱스 구성은?

MAX를 구한 후 MAIN에 제공하는 순서를 생각하였지만..

```
Execution Plan
----------------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      6   FILTER
     44    TABLE ACCESS   GOAL: ANALYZED (BY ROWID) OF 'BPD18D'
     48     INDEX (RANGE SCAN) OF 'BPD18D_EMP_ID' (NON-UNIQUE)
      6    SORT (AGGREGATE)
     44     TABLE ACCESS   GOAL: ANALYZED (BY ROWID) OF 'BPD18D'
     48      INDEX (RANGE SCAN) OF 'BPD18D_EMP_ID' (NON-UNIQUE)
```

SEMI JOIN 방식 선택 오류

- 제공자, 확인자형 SEMI JOIN(SUB-QUERY)의 특징에 맞는 실행계획이 수립되어야 함
  - 실행계획을 확인한 후 JOIN METHOD를 정확히 선택하여 실행할 수 있도록 하라!!!
  - 확인자는 확인자로, 제공자는 제공자로...

```
SELECT YYMM ,CLOSE_FLAG ,NVL(PAY_2001_CNT,0)
      ,NVL(PAY_2002_CNT,0)
      ,NVL(PAY_2003_CNT,0) ,NVL(PAY_2004_CNT,0)
      ,NVL(PAY_2004_SAT,0) ,
FROM  DUTY.BPD18D
WHERE (YYMM, EMP_ID, CLOSE_FLAG) =
    ( SELECT :B1, :B2, MAX(CLOSE_FLAG)
        FROM  DUTY.BPD18D
       WHERE  EMP_ID = :B2
         AND  YYMM = :B1 )
```

- 만약 YYMM +EMP_ID + CLOSE_FLAG가 존재한다면?

```
SELECT /*+ INDEX_DESC(INDEX_NAME)*/
            YYMM ,CLOSE_FLAG
           ,NVL(PAY_2001_CNT,0)
           ,NVL(PAY_2002_CNT,0)
           ,NVL(PAY_2003_CNT,0) ,NVL(PAY_2004_CNT,0)
           ,NVL(PAY_2004_SAT,0)
FROM  DUTY.BPD18D
WHERE  EMP_ID = :B2
  AND  YYMM = :B1
  AND  ROWNUM =1
```

GROUP 비용 과다

상황	● 불필요한 집합의 Grouping 으로 비용 발생 ● 필요한 집합 만 Grouping 하자

비효율 원인	개선방안
• 합계 위해 같은 테이블 반복 read	• Copy T , rollup, cube 활용
• 무분별한 null 체크 함수 사용(nvl)	• 그룹핑 시에는 null 체크 제거
• Group by 함수사용 시 WHERE절에서 추출된 ROW수 만큼 FUNCTION이 수행되는 비효율 발생	• 그룹핑 후 함수 적용
• Min, max 검색 비효율	• 인덱스 와 stop key 활용

집합 반복 ACCESS

- 합계를 위해 반복적으로 같은 테이블을 ACCESS
  - 데이터 복제(COPY_T, CONNECT BY LEVEL, ETC)나 ROLLUP, CUBE를 사용

S_EMP를 두 번 ACCESS

```
SELECT  TITLE , SUM(SALARY)
  FROM S_EMP
WHERE DEPT_ID IN (41,42,43)
GROUP BY  TITLE
UNION ALL
SELECT '합계' TITLE , SUM(SALARY)
 FROM S_EMP
WHERE DEPT_ID IN (41,42,43)
```

COPY_T로 바꾸어도 무방

```
SELECT  DECODE(D.NO,1,TITLE,'합계'),
        SUM(SAL)
FROM (SELECT   TITLE, SUM(SALARY) SAL
        FROM S_EMP
       WHERE DEPT_ID IN (41,42,43)
       GROUP BY TITLE ) E,
      (SELECT LEVEL NO
       FROM DUAL
       CONNECT BY LEVEL <=2) D
GROUP BY DECODE(D.NO,1,TITLE,'합계')
```

ROLLUP

```
SELECT
   DECODE(GROUPING(TITLE),1,'합계',TITLE)
    , SUM(SALARY)
FROM   S_EMP
WHERE DEPT_ID IN (41,42,43)
GROUP BY ROLLUP (TITLE)
```

무분별한 NULL 함수 사용

- grouping 함수(sum,count,avg...)는 null data는 제외하고 연산
  - 불필요한 null 제거 함수 남용을 자제하라~~~

NULL_TEST DATA 수 X 2 만큼 NVL 함수 수행

```
SELECT  OWN
      , SUM( NVL(SAL1,0) ) - SUM( NVL(SAL2,0) )
FROM NULL_TEST
GROUP BY OWN
```

GROUP BY 결과 로우마다
NVL 함수 2 회씩만 수행

```
SELECT   OWN
       , NVL(SUM(SAL1),0) - NVL(SUM(SAL2),0)
FROM   NULL_TEST
GROUP BY OWN
```

Group by 절에 사용된 function의 부하

- grouping 결과값 만큼이 아닌 전체 해당 row수 만큼 function 수행됨
  - 최종 결과에 Function사용을 권장

```
CREATE OR REPLACE FUNCTION F_GET_DNAME (v_dept_id NUMBER )
RETURN VARCHAR2 IS
D_NAME  VARCHAR2(20);
BEGIN
        SELECT NAME INTO D_NAME
        FROM S_DEPT
        WHERE DEPT_ID = v_dept_id;
RETURN d_name;
END;
```

부서코드 입력 시
부서명 리턴

```
SELECT  F_GET_DNAME(DEPT_ID)
        , SUM(SALARY)
FROM    S_EMP
GROUP BY F_GET_DNAME(DEPT_ID)
```

전체 rows =25
결과 rows =12

몇 번 실행?

Group by 절에 사용된 function의 부하

함수안의 SQL 이 수행됨

전체 ROWS수만큼
SQL이 수행됨

```
SELECT NAME FROM S_DEPT
WHERE DEPT_ID = :b1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          1          0           0
Execute     25      0.00       0.00          0          0          0           0
Fetch       25      0.00       0.00          0         50          0          25
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       51      0.00       0.00          0         51          0          25

Parsing user id: 341  (ESI030)  (recursive depth: 1)

Rows     Row Source Operation
-------  ---------------------------------------------------
     25  TABLE ACCESS BY INDEX ROWID S_DEPT
     25   INDEX UNIQUE SCAN S_DEPT_ID_PK (object id 130402)
```

Main에 의해 호출된
SQL이 수행될 경우

Group by 절에 사용된 function의 부하

```
SELECT   F_GET_DNAME(DEPT_ID), SAL
FROM   ( SELECT   DEPT_ID , SUM(SALARY) SAL
           FROM   S_EMP
          GROUP BY DEPT_ID  )
```

```
select f_get_dname(dept_id),
       sum(salary)
from   s_emp
group by dept_id
```

```
SELECT NAME FROM S_DEPT
WHERE DEPT_ID = :b1
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.00	0	0	0	0
Execute	12	0.00	0.00	0	0	0	0
Fetch	12	0.00	0.00	0	24	0	12
total	25	0.00	0.00	0	24	0	12

최종 Grouping 결과 수
만큼 Function수행됨

Group by 절에 사용된 function의 부하

- 사용자 함수는 공유 가능한 오브젝트이므로 적절하게 사용하면 성능 향상에 도움이 되지만, Call이 발생할 때마다 SQL이 실행되는 구조이므로 주의해서 적용해야 한다.

아래와 같은 결과가 나타난 원인에 대해 설명해 보시오.

```
SELECT BRCH_CD, F_CODE(:S01, BRCH_CD) BRCH_NM, TEAM_CD, F_CODE(:S02, TEAM_CD) TEAM_NM, CHP,
       F_GETEMPNAME(CHP) CHP_NM, SUM( DECODE( A.TEAM_CD, :S03, NVL( A.TOT_DLQ_AMT, :N01 ), ……
 FROM TD7B01 A, TD7F03 B
WHERE A.MGT_YM = B.MGT_YM AND A.CUST_CLAS_CD = B.CUST_CLAS_CD AND A.STL_UNT_N = B.STL_UNT_N
   AND  B.STD_D = :S09 AND A.MGT_YM = :S10 AND A.CHP = :S11 AND A.BRCH_CD = :S12 AND A.TEAM_CD = :S13
GROUP BY BRCH_CD, F_CODE(:S14, BRCH_CD), TEAM_CD, F_CODE(:S15, TEAM_CD), CHP, F_GETEMPNAME(CHP)
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.02	1.15	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	1	1.13	8.28	891	2928	0	1
total	3	1.15	8.30	891	2928	0	1

8.30

Rows	Execution Plan
0	SELECT STATEMENT Hint=CHOOSE
1	SORT GROUP BY
611	NESTED LOOPS
612	TABLE ACCESS BY INDEX ROWID TD7B01
612	INDEX RANGE SCAN AK_TD7B01_3
611	TABLE ACCESS BY INDEX ROWID TD7F03
1222	INDEX UNIQUE SCAN PK_TD7F03

AK_TD7B01_3 : MGT_YM + BRCH_CD + TEAM_CD + CHP
PK_TD7F03 : MGT_YM + STL_UNT_N + CUST_CLAS_CD + STD_D
많지 않은 데이터인데 8초? CPU와 ELAPSED의 차이가 7초?

MIN/MAX검색 비효율

- 집합의 MIN/MAX를 얻기 위한 GROUPING 비효율
  - 인덱스와 STOP KEY를 활용하여 MIN/MAX의 부하 최소화

```
SELECT MAX(buy_itemamt)
  FROM buyitem
 WHERE buy_no IN
              (SELECT MAX(buy_no)
                 FROM buyitem
                WHERE itemcd = :b1
                  AND storage_cd = :b2)
   AND itemcd = :b1
   AND storage_cd = :b2
```

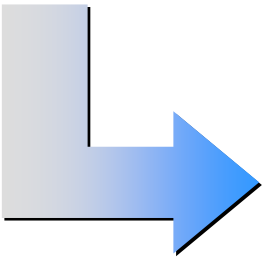
INDEX_01 : itemcd + storage_cd + buy_no + buy_itemamt

MIN/MAX검색 비효율

- 집합의 MIN/MAX를 얻기 위한 GROUPING 비효율
  - 인덱스와 STOP KEY를 활용하여 MIN/MAX의 부하 최소화

```
SELECT MAX(buy_itemamt)
  FROM buyitem
 WHERE buy_no IN
              (SELECT MAX(buy_no)
                 FROM buyitem
                WHERE itemcd = :b1
                  AND storage_cd = :b2)
   AND itemcd = :b1
   AND storage_cd = :b2
```

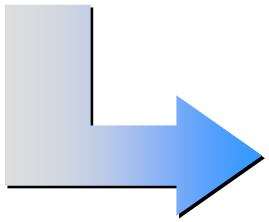
INDEX_01 : itemcd + storage_cd + buy_no + buy_itemamt



```
SELECT /*+ INDEX_DESC(buyitem, INDEX_01) */
          buy_itemamt
  FROM buyitem
WHERE itemcd      = :b1
   AND storage_cd = :b2
   AND ROWNUM     = 1
```

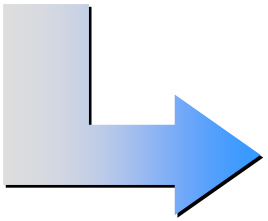
MIN/MAX검색 비효율

```
SELECT LPAD(MAX(INTERVAL), 3, '0')
       ||LPAD(MIN(TIME), 3, '0')
       ||LPAD(MAX(TIME), 3, '0') INTERVAL
FROM  작업일정
WHERE CMP_ID = 'ABC'
AND   START_DT = (SELECT MAX(START_DT)
                  FROM 작업일정
                  WHERE CMP_ID = 'ABC'
                    AND START_DT<='20060308')

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  SORT AGGREGATE
     35   INDEX FAST FULL SCAN 작업일정_PK (UNIQUE)
      2    SORT AGGREGATE
     35     INDEX FAST FULL SCAN 작업일정_PK (UNIQUE)
```



MIN/MAX검색 비효율

```
SELECT LPAD(MAX(INTERVAL), 3, '0')
       ||LPAD(MIN(TIME), 3, '0')
       ||LPAD(MAX(TIME), 3, '0') INTERVAL
FROM  작업일정
WHERE CMP_ID = 'ABC'
AND   START_DT = (SELECT MAX(START_DT)
                    FROM 작업일정
                   WHERE CMP_ID = 'ABC'
                     AND START_DT<='20060308')

Rows     Row Source Operation
-------  ---------------------------------------------
      1  SORT AGGREGATE
     35   INDEX FAST FULL SCAN 작업일정_PK (UNIQUE)
      2    SORT AGGREGATE
     35     INDEX FAST FULL SCAN 작업일정_PK (UNIQUE)
```



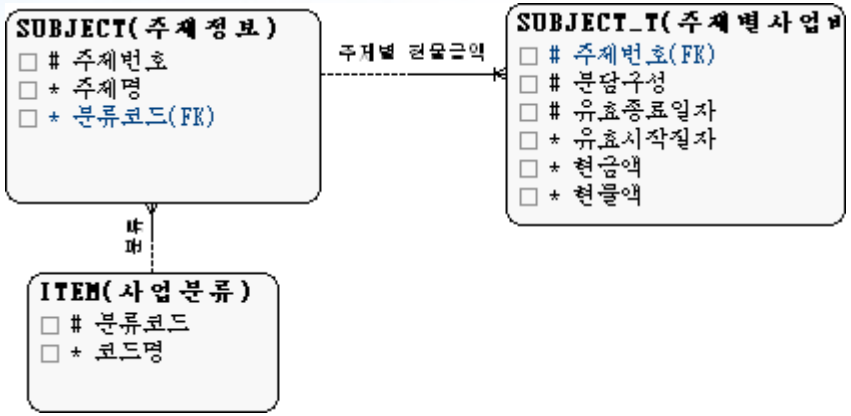
```
SELECT /*+ INDEX(A 작업일정_PK) */
         LPAD(MAX(INTERVAL), 3, '0')
       ||LPAD(MIN(TIME), 3, '0')
       ||LPAD(MAX(TIME), 3, '0') INTERVAL
 FROM  작업일정 A
 WHERE CMP_ID = 'ABC'
 AND   START_DT =
         (SELECT /*+ INDEX_DESC(X 작업일정_PK) */
                 START_DT
          FROM 작업일정 X
          WHERE CMP_ID = 'ABC'
            AND START_DT <= '20060308'
            AND ROWNUM = 1 )

Rows     Row Source Operation
-------  ---------------------------------------------
      1  SORT AGGREGATE
     35   INDEX RANGE SCAN 작업일정_PK (UNIQUE)
      2    COUNT STOPKEY
      1     INDEX RANGE SCAN DESCENDING 작업일정_PK
```

컬럼추가로 인한 부하

상황	컬럼 변형 및 컬럼 추가 시 비용 과다
비효율 원인	개선방안
로우를 컬럼으로 변환 시 다수 쿼리 이용	Sum(decode)활용
전,후,등위,누적 컬럼 추가 루프 로직 사용	Analytic 함수 사용
다수의 함수 사용	단일 함수의 파라미터 분할 방법 강구
인라인 뷰에서 함수관련 컬럼은 메인 쿼리에서 두 번 실행	인라인 뷰에서 같은 레벨의 group by 적용
자식의 집계 컬럼 추가 시 전체범위 처리	집계 컬럼을 함수 혹은 스칼라 서브 쿼리 사용

컬럼 변형 및 추가 시 비용 발생

- ROW → 컬럼으로 변형을 위해 동일 집합을 반복적으로 ACCESS하는 다수 SQL활용
  : SUM(DECODE), MAX(DECODE), SUM(CASE)와 같은 집합연산자를 활용



- 주제별사업비분담.분담구성
  101, 104, 105 : 출연금
  103 : 민간부담금
  107 : 기타

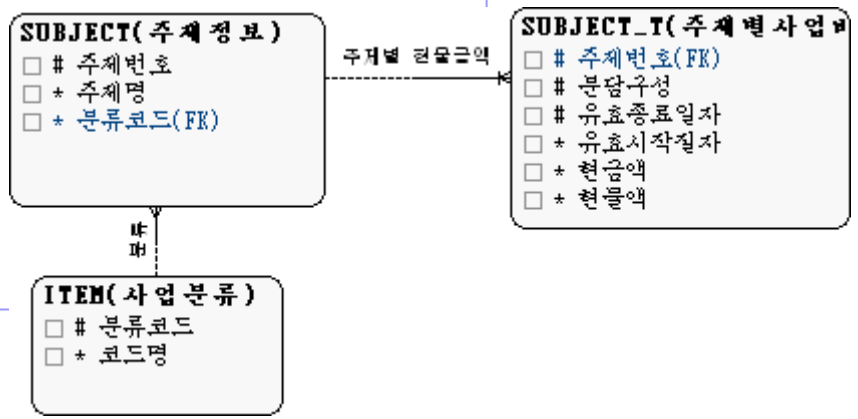
```
SELECT  S.주제번호, S.주제명, S.분류코드, B.코드명,   A1.MNY "출연현금",
        A2.MNY "출연현물",   A3.MNY "민간현금",   A4.MNY "민간현물"
FROM   SUBJECT S
    , ITEM B
    , (SELECT A.주제번호, SUM(현금액) MNY FROM SUBJECT_T A
       WHERE A.주제번호 = :주제번호
         AND A.분담구성 IN ('101', '104', '105')
         AND A.유효종료일 = '99991231'
       GROUP BY A.주제번호) A1
    , (SELECT A.주제번호, SUM(현물액) MNY FROM SUBJECT_T A
       WHERE A.주제번호 = :주제번호
         AND A.분담구성 IN ('101', '104', '105')
         AND A.유효종료일 = '99991231'
       GROUP BY A.주제번호) A2
    , (SELECT A.주제번호, SUM(현금액) MNY FROM SUBJECT_T A
       WHERE A.주제번호 = :주제번호
         AND A.분담구성 = '103'
         AND A.유효종료일 = '99991231'
       GROUP BY A.주제번호) A3
    , (SELECT A.주제번호, SUM(현물액) MNY FROM SUBJECT_T A
       WHERE A.주제번호 = :주제번호
         AND A.분담구성 = '103'
         AND A.유효종료일 = '99991231'
       GROUP BY A.주제번호) A4
WHERE S.주제번호 = :주제번호
  AND B.분류코드 = S.분류코드
  AND A1.주제번호 = S.주제번호 AND A2.주제번호 = S.주제번호
  AND A3.주제번호 = S.주제번호 AND A4.주제번호 = S.주제번호
```

동일집합 반복 ACCESS

컬럼 변형 및 추가 시 비용 발생

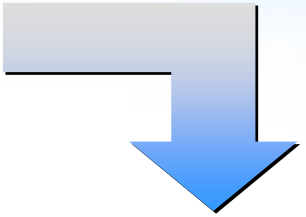
- 논리적 통분을 통해 한번만 access하고 종료
- 각 집합 별로 별도의 가공작업이 필요할 수 있음.
- DECODE를 CASE로?

```
select    s.주제번호, s.주제명, s.분류코드, b.코드명,
          sum(decode(a.분담구성, '101', 현금액, '104', 현금액, '105', 현금액)) "출연현금",
          sum(decode(a.분담구성, '101', 현물액, '104', 현물액, '105', 현물액)) "출연현물",
          sum(decode(a.분담구성, '103', 현금액)) "민간현금",
          sum(decode(a.분담구성, '103', 현물액)) "민간현물"
from      subject s
          , item b
          , subject_t a
where s.주제번호 = :주제번호
   and b.분류코드 = s.분류코드
   and a.주제번호 = s.주제번호
   and a.유효종료일 = '99991231'
Group by s.주제번호, s.주제명, s.분류코드, b.코드명
```



컬럼 변형 및 추가 시 비용 발생

```
SELECT (SELECT COUNT (USER_ID) FROM MAILBOX
        WHERE USER_ID = :V_USER_ID    ) TOTALCOUNT,
       (SELECT COUNT (USER_ID) FROM MAILBOX
        WHERE USER_ID = :V_USER_ID
       AND (   STATUS = 'SENT'     OR STATUS = 'DELIVERED'
  OR STATUS = 'SENTFAIL' OR STATUS = 'SENTCANCEL')) SENTCOUNT,
       (SELECT COUNT (USER_ID) FROM MAILBOX
        WHERE USER_ID    = :V_USER_ID
          AND STATUS     = 'SENT'
          AND DELVTIME  IS NOT NULL
          AND READTIME  IS NULL       ) UNSEENCOUNT,
       (SELECT COUNT (USER_ID) FROM MAILBOX
        WHERE USER_ID      = :V_USER_ID
          AND READTIME  IS NOT NULL
          AND STATUS     = 'SENT'     ) SEENCOUNT,
       (SELECT COUNT (USER_ID) FROM MAILBOX
        WHERE USER_ID = :V_USER_ID
          AND (   STATUS = 'SENTCANCEL'
        OR STATUS = 'RESERVECANCEL')  ) CANCELCOUNT
  FROM DUAL
```



```
SELECT COUNT(*) TOTALCOUNT
      ,COUNT(CASE WHEN STATUS IN
('SENT','DELIVERED','SENTFAIL','SENTCANCEL')
                  THEN 1
             END ) SENTCOUNT
      ,COUNT(CASE WHEN STATUS = 'SENT'
                   AND DELVTIME IS NOT NULL
                   AND READTIME IS NULL THEN 1
             END ) UNSEENCOUNT
      ,COUNT(CASE WHEN READTIME IS NOT NULL
                   AND STATUS = 'SENT' THEN 1
             END ) SEENCOUNT
      ,COUNT(CASE WHEN STATUS IN ('SENTCANCEL','RESERVECANCEL')
                  THEN 1
             END ) CANCELCOUNT
  FROM MAILBOX A
 WHERE A.USER_ID = :V_USER_ID
```

자식의 집계 컬럼 추가 시 전체범위처리

- 1 : M 관계에서 M의 값에 대한 집계처리 시 전체 범위 처리 불가피 → 함수나 스칼라 서브쿼리를 사용하여 부분범위처리로 유도

```
SELECT x.사번, x.성명, x.직급, x.직책, …,
       AVG(y.급여총액) 평균급여
FROM   사원 x,  급여 y
WHERE x.사번 = y.사번
       and  x.부서  = '1110'
       and  y.급여년월 between '199801'
                  and        '199807'
GROUP BY x.사번, x.성명, x.직급, x.직책, …
```

Function 사용

```
CREATE OR REPLACE FUNCTION F_GET_AVG ( V_사번
NUMBER
                                , V_ST VARCHAR2
                                , V_ED VARCHAR2)
RETURN NUMBER IS
AVG_급여총액   VARCHAR2(20);
BEGIN
    SELECT AVG(급여총액) INTO AVG_급여총액
      FROM  S_EMP
     WHERE 사번 = V_사번
      AND  급여년월 BETWEEN V_ST AND V_ED ;
RETURN AVG_급여총액;
END;
```

```
SELECT   X.사번, X.성명, X.직급, X.직책
       , F_GET_AVG(X.사번, :V_ST,:V_ED)
FROM   사원  X
WHERE  X.부서 = '1110'
```

스칼라 서브쿼리 사용

```
SELECT   X.사번 , X.성명 , X.직급 , X.직책
       , (SELECT AVG(급여총액)
          FROM  급여 Y
          WHERE Y.급여년월 BETWEEN '199801'
                              AND '199807'
          AND   X.사번 = Y.사번 ) 평균급여
FROM   사원  X
WHERE  X.부서 = '1110'
```

요소 별 영향도의 이해

- APP 개발형태 영향요소

Literal SQL

- OLTP 환경에서 Literal SQL의 남용은 심각한 Parse 부하를 유발하여 시스템 성능저하의 주요 원인을 제공
- Static SQL로 전환할 수 없는 시스템의 특징은?

```
SELECT A.SCRBR_NO, A.APLN_NM, A.APLN_TEL_NO, ……
  FROM DA01T01 A, CC01T01 B, DB04T01 C, CC04T01 D, BB01T01 E, CB01T01 F
 WHERE A.SCRBR_NO    = B.SCRBR_NO
   AND A.AS_RECV_NO  = C.AS_RECV_NO
   AND A.SCRBR_NO    = D.SCRBR_NO
   AND A.SCRBR_NO    = E.SCRBR_NO
   AND E.CHG_KIND_CD = '101'
   AND E.RECV_NO     = F.RECV_NO
   AND D.SEQ_NO      = 1
   AND A.INSTAL_배송지코드 = 'L10277'
   AND A.AS_WK_STAT_CD     = '01'
 ORDER BY C.AS_RECV_NO DESC
```

심각한 HARD PARSING 부하 !!

call	count	cpu	elapsed	disk	query	current	rows
Parse	20	0.81	1848.30	0	0	0	0
Execute	20	0.01	0.01	0	0	0	0
Fetch	20	0.74	1.58	114	15693	0	50
total	60	1.56	1849.89	114	15693	0	50

순환전개 비효율

- 잘못 사용된 인덱스 힌트로 인해 순환전개에 성능 저하 발생
- 오래된 통계정보로 정확한 실행계획 수립이 안됨

```
SELECT DISTINCT B.CRP_STL_N, C.STL_CL, C.BLL_MTD_CD
FROM ( SELECT /*+ INDEX(TC5002, PK_TC5002)*/
                  LPAD( :S001, :N001 * ( LEVEL - :N002 ) ) || ACT_HG_NM ACT_HG_NM, CRP_ACT_N …
         FROM TC5002
          START WITH CRP_ACT_N = :S004 CONNECT BY PRIOR CRP_ACT_N = UP_ACT_N ) A,
        TC5008 B, TC5003 C
WHERE A.CRP_ACT_N = B.CRP_ACT_N AND B.CRP_STL_N = C.CRP_STL_N
  AND A.CRP_ACT_LVL_CD = :S005
```

```
0  SELECT STATEMENT   GOAL: CHOOSE
      1   SORT (UNIQUE)
      1    NESTED LOOPS
      2     HASH JOIN
      1      VIEW
      1       FILTER
      2        CONNECT BY
      2         INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'PK_TC5002' (UNIQUE)
      1         TABLE ACCESS   GOAL: ANALYZED (BY USER ROWID) OF 'TC5002'
      1         TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TC5002'
  13906          INDEX   GOAL: ANALYZED (FULL SCAN) OF 'PK_TC5002' (UNIQUE)
 160087       TABLE ACCESS   GOAL: ANALYZED (FULL) OF 'TC5008'
      1     TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'TC5003'
      2      INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'PK_TC5003' (UNIQUE)
```

- 순환전개시에 잘못된 hint의 사용으로 인해 PK_TC5002(CRP_ACT_N)을 선두로 하는 인덱스를 사용하도록 지정하였으므로 INDEX FULL SCAN발생.
- 전개시에 UP_ACT_N를 선두로 하는 인덱스가 있음에도 불구하고 사용하지 못함. TC5008 조인시 CRP_ACT_N (법인개정번호)를 선두로 하는 인덱스가 있으나 **오래된 통계정보의 사용**으로 인해 FULL SCAN처리하며 HASH JOIN처리됨

SQL 작성 형태의 영향

- 불필요한 ACCESS 효율화
- 단순 복사 -> 붙여 넣기의 폐해
- 이런 유형의 SQL이 발생한 경우를 유추해봅시다.

아래와 같은 결과가 나타난 원인에 대해 설명해 보시오.

```
select count( * )
from ( select rownum as rnum, rec_key, title_info,  author_info, pub_info, pub_year_info, mat_code,
              use_obj_code, form_code, media_code, place_info, working_status,
              contents_yn, abstracts_yn, wonmun_yn
       from ( select a.rec_key, a.title_info, a.author_info, a.pub_info, a.pub_year_info, a.mat_code,
                     a.use_obj_code, a.form_code, a.media_code,
                     ( select max( c.description )
                         from cd_code_tbl c
                        where c.class = '19'
                          and trim( c.code ) = trim( a.main_place_info ) ) as place_info,
                     a.working_status, a.contents_yn, a.abstracts_yn,
                     a.wonmun_yn
                from IDX_BO_TBL a
               where a.working_status not in ( 'BOT211O', 'BOT212O' )
                 and a.rec_key in ( select eco_key
                                      from ECO$V_BOIDXALLITEM$I
                                     where TOKEN = '신기한열매' ) ) )
```

SQL 작성 형태의 영향

call	count	cpu	elapsed	disk	query	current	rows
Parse	11	0.48	2969.13	0	0	22	0
Execute	11	0.02	10.50	0	0	0	0
Fetch	11	0.00	0.02	0	33	0	11
total	33	0.50	2979.65	0	33	22	11

```
Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT   GOAL: CHOOSE
      1   SORT (AGGREGATE)
      1    VIEW
      1     COUNT
      1      NESTED LOOPS
      2       VIEW OF 'VW_NSO_1'
      2        SORT (UNIQUE)
      1         REMOTE [UP2ES]
                   SELECT "ECO_KEY","TOKEN" FROM "ECO$V_BOIDXALLITEM$I"
                     "ECO$V_BOIDXALLITEM$I" WHERE "TOKEN"='신기한열매'
      1       TABLE ACCESS   GOAL: ANALYZED (BY INDEX ROWID) OF 'IDX_BO_TBL'
      2        INDEX   GOAL: ANALYZED (UNIQUE SCAN) OF 'IDX_BO_TBL_PK' (UNIQUE)
```

SQL 작성 형태의 영향 - 개선안

```
select count(a.rec_key)
 from IDX_BO_TBL a
where a.working_status not in ( 'BOT2110', 'BOT2120' )
   and a.rec_key in ( select eco_key
                        from ECO$V_BOIDXALLITEM$I
                       where TOKEN = '신기한열매' )
```

- 읽어야 하는 범위만 읽고 count를 하여 비효율을 줄일 수 있었다.
- 내가 작성한 SQL에 대해 책임질 수 있어야 함.

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.01	0	2	0	0
Execute	1	0.00	0.02	0	0	0	0
Fetch	1	0.00	0.01	0	3	0	1
Total	3	0.01	0.04	0	5	0	1

```
Rows     Row Source Operation
----------  -------------------------------------------------
         1  SORT AGGREGATE
         1   NESTED LOOPS
         2    VIEW VW_NSO_1
         2     SORT UNIQUE
         1      REMOTE
         1    TABLE ACCESS BY INDEX ROWID IDX_BO_TBL
         2     INDEX UNIQUE SCAN (IDX_BO_TBL_PK)
```

영향요소 정리

Literal SQL(Dynamic)	❑ 심각한 Parsing 부하 ❑ Shared Memory 사용 효율 저하	→	❑ Static SQL 사용 ❑ Bind Variable 사용
OPTIMIZING 전략 부재	❑ 빈번한 Full Table Scan ❑ 비효율적인 엑세스	→	❑ 신규/변경 인덱스 생성 및 SQL 최적화
SQL 사용 오류	❑ 인덱스 컬럼 가공 ❑ 과도한 DBMS CALL ❑ 비즈니스 이해부족,SQL 사용 오류	→	❑ 인덱스 컬럼 가공 금지 ❑ SQL 통합 및 힌트 사용으로 액세스 효율 향상 ❑ 비즈니스를 반영하여 재 작성
프로그램 구조 비효율	❑ 부적절한 기능설계 ❑ 절차형 처리에 의존	→	❑ 기능 재설계 ❑ 통합 SQL 사용으로 비절차형 프로그램화
물리DB설계 오류	❑ 부적절한 데이터 타입 ❑ LONG 타입 사용 ❑ 과도한 Chain 발생 ❑ 스토리지 파라미터 옵션	→	❑ 적절한 데이터 타입 ❑ LOB 타입 사용 ❑ Reorganization를 통한 해결
논리DB설계 오류	❑ Data Integrity / Constraint 위반 ❑ 정규화 위반 ❑ ERD 부재	→	❑ 적절한 제약조건 추가 ❑ 정규화 작업 ❑ ERD 작성
부적절한 LOCK LEVEL	❑ 과도한 Locking Level ❑ 과도한 Locking 지속 시간	→	❑ 적절한 Locking Level 사용 ❑ Locking 유지 시간 최소화
응용 프로그램 성능 개선	❑ Bind Variable 기법 미 사용 ❑ Array Fetch 기법 미 적용 ❑ 전체범위 처리 ❑ 디버그 버전 사용	→	❑ Bind Variable 기법 적용 ❑ Array Fetch 기법 적용 ❑ 부분범위 처리 기법 적용 ❑ 릴리스 버전 사용